A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Tesina Finale di

Algoritmi e Strutture Dati

Corso di Laurea in Ingegneria Informatica ed Elettronica, Curriculum Informatica

A.A. 2022-2023

Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

Implementazione dell'algoritmo di Dijkstra

studente

329118 **Francesco Mancinelli** francesco.mancinelli3@studenti.unipg.it

Data ultimo aggiornamento: 16 giugno 2023

0. Indice

1. Introduzione

Lo scopo del progetto consiste nell'implementare l'algoritmo di Dijkstra per il calcolo dei cammini minimi da sorgente unica e nell'effettuare una valutazione sperimentale delle sue prestazioni per varie tipoligie di grafi di input, ipotizzando la completa connessione.[6] Per poter spiegare in dettaglio l'implementazione sarà prima necessario definire alcune nozioni base.[9] Un grafo *G = (V,E)* è una struttura composta da nodi (o vertici) V e archi E. Gli archi sono rappresentati con delle coppie di valori $(u, v) \in E$. Corrisponde all'arco che connette il nodo u al nodo v. Inoltre ad ogni arco può essere assegnato un peso, ovvero il costo necessario per spostarsi dal nodo di partenza al nodo di arrivo. Se agli archi viene associato un peso, allora il grafo si dirà pesato. Inoltre possiamo denotare con *deg(v)*, il grado di un vertice v, ovvero il numero di archi che coincide su di esso. Nel caso in cui il vertice sia isolato il grado sarà pari a zero. Un grafo G è detto connesso se, per ogni coppia di vertici $(u, v) \in V$, esiste un cammino che collega u a v. Un grafo può essere non orientato oppure orientato. Nel primo caso è un insieme contenente coppie non ordinate *(u,v)*, mentre nel secondo caso contiene coppie orientate *(u,v)*. [8] [10] Il problema del cammino minimo consiste nel trovare un percorso tra due

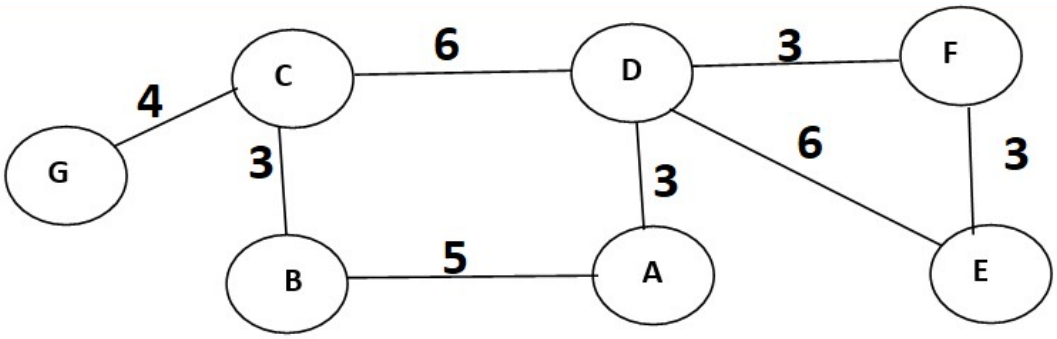

Figura 1.1: Grafo pesato non orientato

nodi (o vertici) in un grafo tale da minimizzare la somma dei pesi dei suoi archi costituenti. Possiamo individuare tre tipi di problemi:

- *Single-Source Shortest Path (SSSP).* Consiste nel trovare i percorsi più brevi tra un dato vertice v e tutti gli altri vertici nel grafico. Algoritmi come *Breath-First-Search (BFS)* per grafici non ponderati o Dijkstra risolvono questo problema.

- *All-Pairs Shortest Path (APSP).* Consiste nel trovare il percorso più breve tra tutte le coppie di vertici nel grafico. Per risolvere questo secondo problema, si può utilizzare l'algoritmo.

- *Single-Pair Shortest Path (SPSP).* Consiste nel trovare il percorso più breve tra una singola coppia di vertici. Questo problema è per lo più risolto utilizzando $A*$, che può considerarsi una versione modificata di Dijkstra.

2. Descrizione del Problema

L'obiettivo del progetto è di implementare l'algoritmo di Dijkstra per il calcolo dei cammini minimi da sorgente unica, ovvero il problema *SSSP*. Dopo aver descritto i principali algoritmi usati nella ricerca dei cammini minimi, ed aver approfondito il funzionamento dell'algoritmo di Dijkstra, si andranno ad effettuare delle valutazioni sperimentali per varie tipologie di grafi in input. I grafi su cui testare l'algoritmo dovranno essere generati casualmente dato un determinato valore di densità, ovvero il rapporto tra il numero di archi e il numero di nodi del grafo. Inoltre dovrà essere testato anche per dimensioni variabili del grafo, così da mostrare l'andamento della sua complessità.

3. Algoritmi e strutture dati analizzati ed implementati

3.1 Liste e Matrici di adiacenza

Per poter implementare il codice dell'algoritmo di Dijkstra ci si è serviti di una lista di adiacenza. L'indice dell'array rappresenta un nodo e ogni elemento nella sua lista rappresenta gli altri nodi che hanno un arco collegato con esso. Ad ogni elemento dell'array è collegata una lista.[7] L'indice dell'elemento dell'array corrisponde ad un nodo, la lista corrisponde ai nodi adiacenti ad esso. Il principale vantaggio della lista di adiacenza è quello di essere efficiente in termine di archiviazione perchè memorizza solo i valori dei nodi adiacenti e non il riferimento a tutti i nodi del grafo. Se il grafo non è denso, ovvero ha molti nodi e pochi archi, è preferibile utilizzare le liste di adiacenza. In alternativa alle liste di adiacenza potrebbe essere utilizzata una matrice di adiacenza, dove, gli indici i e j corrispondono ai nodi.[3] La coordinata $A_{i,j}$ sarà pari ad 1 se sono collegati, 0 in caso contrario. Il principale vantaggio della matrice di adiacenza è il tempo di esecuzione ridotto per le operazioni di aggiunta, rimozione e controllo di un arco, e la facilità di gestione delle matrici. Inoltre è preferibile se il grafo è denso, ovvero ha un elevato numero di archi.

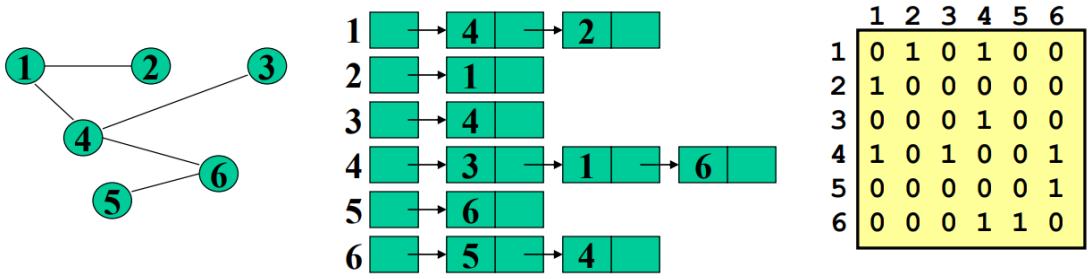

Figura 3.1: Liste di adiacenza al centro, matrici di adiacenza a sinistra

3.2 L'algoritmo di Dijkstra

Questo algoritmo fu inventato nel 1956 dall'informatico olandese Edsger Dijkstr, ed è utilizzato per cercare i cammini minimi di un grafo con o senza ordinamento, connesso e pesato non negativamente.[4] L'algoritmo di Dijkstra era stato originariamente sviluppato per trovare la distanza minima fra due nodi, utilizzando il concetto di coda di priorità. Successivamente ha trovato applicazione anche con gli heap-binari. Le applicazioni di questo algoritmo sono molto ampie: può essere utilizzato per trovare la distanza minima tra due località, per il routing IP, per i protocolli di rete IS-IS e OSPF, per suggerire nuovi amici sui social network, per le reti telefoniche, idriche, stradali, circuitali e molto altro. Questo è un algoritmo goloso in quanto viene scelto sempre il nodo più vicino. É dinamico perchè le distanze vengono aggiornate usando i valori calcolati in precedenza. Si basa sulle proprietà che il sottocammino di un cammino minimo è anch'esso un cammino minimo.

3.2.1 Funzionamento

Scelto un nodo di partenza "source", definiamo con Y la distanza dal nodo di partenza. L'algoritmo di Dijkstra partirà inizialmente con distanze infinite e cercherà di migliorarle passo dopo passo.

1. Come primo passo, crea un insieme con tutti i nodi contrassegnandoli come ancora non visitati.

2. Successivamente, assegna ad ogni nodo una distanza provvisoria. Imposta la distanza del nodo di partenza da se stesso pari a 0, e la distanza degli altri nodi con la sorgente pari ad infinito. Durante l'esecuzione dell'algoritmo, le distanze provvisorie dei nodi rispetto alla sorgente si aggiorneranno, mantenendo in memoria sempre la distanza più breve scoperta fino a quel momento.

3. Partendo dal nodo iniziale, calcola le distanze provvisorie dal nodo corrente ai nodi direttamente collegati. Quindi confronta la distanza provvisoria appena calcolata con quella attualmente assegnata al vicino, così da assegnargli quella più piccola.

4. Dopo aver visitato tutti i nodi collegati al nodo corrente, il nodo corrente viene contrassegnato come visitato rimuovendolo dall'insieme dei nodi non visitati. Un nodo visitato non verrà mai più visitato in quanto qualsiasi visita successiva risulterebbe avere una distanza maggiore.

5. I passi 3 e 4 si ripetono finchè il nodo di destinazione non viene contrassegnato come visitato (nel caso del problema *SPSP*) oppure tutti gli altri nodi dovranno essere contrassegnati come visitati (nel caso del problema *SSSP*), oppure se la minima distanza provvisoria tra i nodi dell'insieme non visitato è infinita (ovvero se non è presente un percorso che collega i due nodi).

3.2.2 Esempio

Dato che l'algoritmo di Dijkstra è goloso, questo sceglierà il percorso più breve per andare da un nodo all'altro.[5] L'algoritmo termina quando si visitano tutti i nodi del grafo raggiungibili dalla sorgente.[10] Tuttavia a volte, quando si trova un nuovo nodo possono esserci percorsi più brevi attraverso di esso da un nodo visitato ad un altro nodo già visitato. Guardando la figura il nodo 0 ha due

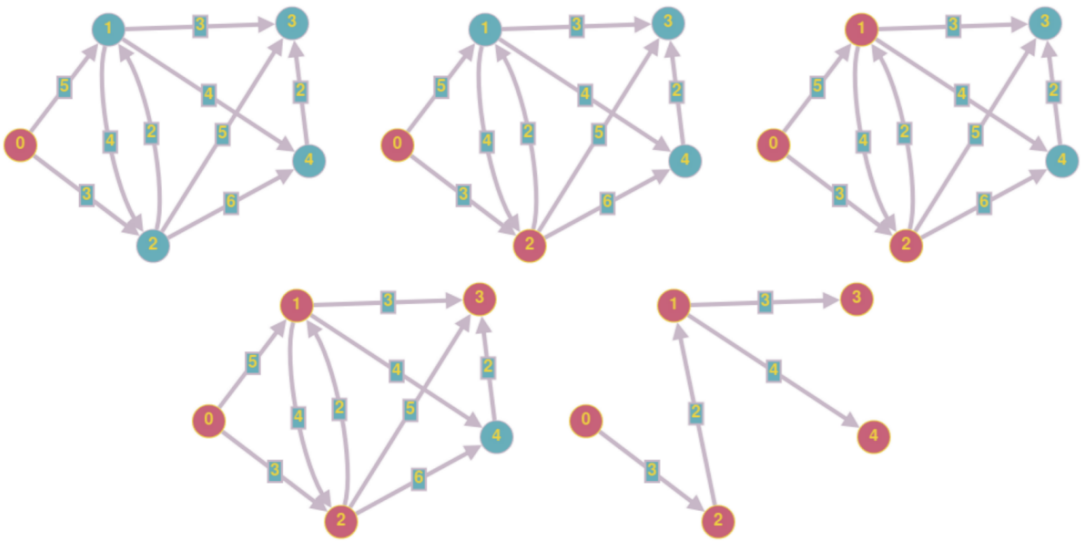

Figura 3.2: Esempio dell'algoritmo di Dijkstra

possibili strade. La prima va da 0 a 1 con costo 5, l'altra da 0 a 2 con costo 3. A questo punto si scriveranno i due nodi nella lista dei nodi come nodi visitati, con anche i costi relativi per raggiungerli. Scegliamo quindi il percorso $0 \rightarrow 2$. Quando visitiamo il nodo 2 si può vedere che si hanno 3 strade. La prima è da 2 a 1, la seconda da 2 a 3, la terza da 2 a 4. Quindi si scrive nella lista i due nuovi nodi e si sceglierà il percorso più breve. In questo caso da 2 a 1. Successivamente in 1 si hanno i cammini da 1 a 2, 3 e 4. Quindi si sceglie il cammino più breve (da 1 a 3). Quindi scriviamo nella lista i nuovi costi dei cammini dal nodo 0 agli altri nodi se non erano presenti. Poi si può vedere che non ci sono più nuovi percorsi da 3 che lo collegano a 4. Quindi si torna al nodo precedente controllando il percorso più breve. C'è un percorso con costo 4 che va a 4 e un percorso che va a 2. Qui viene scelto uno dei due. 'E indifferente in quanto i perocorsi più brevi da 0 a 4 sono scritti nell'elenco. Infine, si possono mostrare tutti i percorsi minimi.

3.2.3 Osservazione e comparazione con A-star

L'algoritmo di Dijkstra non fa alcun tentativo di "esplorazione" diretta verso la destinazione. L'unica considerazione che l'algoritmo fa nel determinare il prossimo vertice da scegliere è la sua distanza dal punto di partenza. Questo significa che l'algoritmo si espande in modo uniforme rispetto al punto di partenza, non considerando in alcun modo dove il nodo di destinazione sia collocato. Sebbene per il problema *SSSP*, non c'è nessun particolare inconveniente in quanto devono essere comunque trovati tutti i percorsi minimi rispetto a tutti i nodi, per il problema

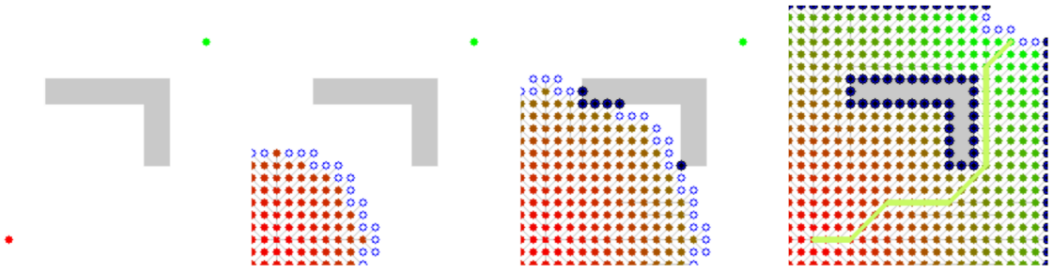

Figura 3.3: Progresso dell'algoritmo di Dijkstra

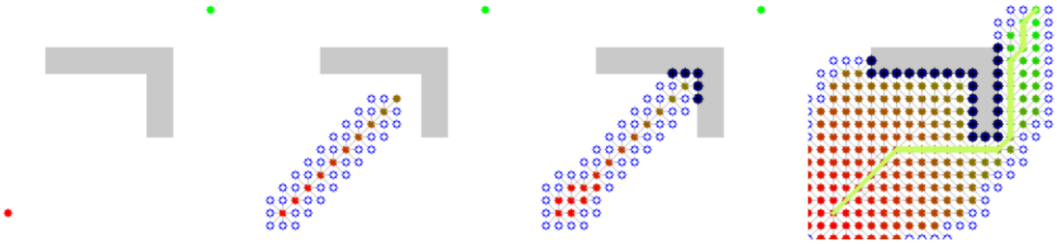

Figura 3.4: Progresso dell'algoritmo A-star

SPSP si potrebbe rivelare piuttosto lento. Comparandolo con $A*$ possiamo notare un notevole spreco di risorse, in quanto vengono visitati molti più nodi.

3.2.4 Pseudocodice

Il primo algoritmo inizializza i valori di d e π ponendo la distanza con ogni nodo a infinito e a null il suo predecessore. Il secondo algoritmo è il processo di rilassamento (RELAX) di un arco *(u,v)* consiste nel verificare se, passando per u, è possibile migliorare il cammino minimo per v e in caso affermativo aggiorna $v.d$ e $v.\pi$. É bene ricordare alcune proprietà del rilassamento:

Algorithm 1: INITIALIZE-SINGLE-SOURCE(G,s)

```
1 for ogni vertice v ∈ G.V do
2 |   v.d = ∞;
3 |   v.π = NIL;
4 end
5 s.d = 0
```

Algorithm 2: RELAX(u,v, w)

```
1 if v.d > u.d + w(u, v) then
2 |   v.d = u.d + w(u, v);
3 |   v.π = u;
4 end
```

- Disuguaglianza triangolare: per qualsiasi arco *(u,v)* $\in E$ si ha $\delta(s, v) \leq \delta(s, v) + w(u, v)$;
- Proprietà del limite superiore: per tutti i nodi *vin*V si ha sempre $v.d \geq \delta(s, v)$ e una volta che $v.d$ diviene pari a $\delta(s, v)$ esso non cambia più;
- Proprietà dell'assenza di un cammino minimo: se non c'è un cammino da s a v allora si ha sempre $v.d = \delta(s, v) = \infty$;
- Proprietà della convergenza: se u è il nodo che precede v in un cammino minimo da s a v e se $u.d = \delta(s, u)$ prima del rilassamento dell'arco *(u,v)* allora $v.d = \delta(s, v)$ dopo il rilassamento di *(u,v)*;
- Proprietà del rilassamento del cammino: se $p = \lfloor v_0, v_1, ..., v_k \rfloor$ è un cammino minimo da $v_0 = s$ a v_k e gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$ allora $v_k.d = \delta(s, v_k)$. Questa proprietà vale indipendentemente da altri passi di rilassamento che vengono effettuati anche se interposti fra il rilassamento degli archi di p.
- Proprietà del sottografo dei predecessori: se G non contiene cicli di peso negativo raggiungibili da s, una volta che $v.d = \delta(s, v)$ per ogni *vin*V, il sottografo dei predecessori è un albero di cammini minimi radicato in s.

Dopo aver analizzato i valori con il metodo INITIALIZE-SINGLE-SOURCE, e l'insieme S pari all'insieme nullo, l'algoritmo mantiene la condizione di invarianza $Q = V - S$ all'inizio di ogni iterazione del while. Poi viene analizzata la lista di priorità con tutti i nodi V. Quindi il nodo u conterrà il più corto percorso stimato di ogni nodo $V-S$. Successivamente, viene eseguito il metodo RELAX aggiornando il valore di $v.d$ e $v.\pi$. Se viene trovato un percorso migliore, questo sarà memorizzato in u. Il loop si ripete esattamente $|V|$ volte, in quanto ogni nodo viene visitato una ed una sola volta. Dijkstra sceglie sempre il nodo più vicino essendo un algoritmo goloso. Le strategie golose non hanno sempre risultati ottimali, ma in questo caso può essere dimostrato che l'algoritmo di Dijkstra restituisce sempre i percorsi più brevi.

Algorithm 3: DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G,s);
2  S = ∅;
3  Q = G.V;
4  while Q ≠ ∅ do
5  |  u =EXTRACT-MIN(Q);
6  |  S = S ∪ {u};
7  |  for ogni vertice v ∈ G.Adj[u] do
8  |  |  RELAX(u,v,w);
9  |  |  DECREASE-KEY(Q,v,v.d);
10 |  end
11 end
```

3.2.5 Correttezza

La correttezza può essere dimostrata per induzione. Si definisce innanzitutto l'ipotesi invariante del ciclo: Per ogni nodo x, la distanza più breve dalla sorgente a x corrisponde a $x.d$ (quando esiste un percorso di nodi visitati che li collegano). Se non esistesse un percorso che collega i due nodi allora la distanza sarà infinito. Successivamente si definisce il caso base, quando $S=$, ovvero non è stato visitato alcun nodo. In questo caso l'ipotesi è banale. Altrimenti, assumendo l'ipotesi che *n-1* nodi sono stati visitati, si sceglie un arco dalla sorgente a x dove si ha il minimo valore di $x.d$ di qualsiasi nodo non visitato y tale che $x.d = y.d$+*S.Edge(x,y)*. Se ci fosse stato un percorso più breve a $x.d$, e se fosse stato il primo nodo non visitato su quel percorso, allora secondo l'ipotesi originale si creerebbe una contraddizione: $w.d > x.d$. Allo stesso modo se esisterebbe un percorso più breve per x senza utilizzare nodi non visitati, e se w fosse il penultimo nodo su quel percorso, allora $x.d = w.d$+*S.Edge(w,x)*, che è una contraddizione. Dopo aver elaborato x, sarà ancora vero che per ogni nodo non visitato w, $d.w$ corrisponderà alla distanza minima tra il nodo di partenza e w usando solamente nodi visitati. Questo perchè se ci fosse stato un percorso più breve, questo non sarebbe passato per x, altrimenti lo avremmo trovato precedentemente e lo avremmo aggiornato durante l'elaborazione del nodo. Quando tutti i nodi sono stati visitati, il percorso più breve dalla sorgente al nodo y consiste di soli nodi visitati, quindi $y.d$ corrisponderà alla distanza più breve.

4. Analisi della Complessità

A livello teorico la complessità dell'algoritmo di Dijkstra può essere rappresentata in funzione del numero dei nodi e del numero di archi. Inoltre essa dipende dal tipo di implementazione della coda di priorità: con Arrays, heap-binari, oppure con un heap di Fibonacci. Le operazioni effettuate sulla coda di priorità sono:

- costruzione della coda;
- rimozione dell'elemento minimo (EXTRACT-MIN);
- riduzione del valore di un elemento (DECREASE-KEY).

La costruzione della coda ha un costo pari a $\theta(|V|)$, per ogni tipo di coda di priorità. L'algoritmo chiama INSERT e EXTRACT-MIN una volta per ogni vertice. Dato che ogni vertice $u \in V$ è aggiunto all'insieme S solamente una volta, ogni arco nella lista di adiacenza è esaminato nel ciclo *for* una volta. Dato che il numero totale di nodi nelle liste di adiacenza sono $|E|$, il ciclo *for* viene eseguito $|E|$ volte. Considerando che la coda delle priorità sia stata realizzata con un array, allora si memorizzerà $v.d$ nel v-esimo elemento dell'array. L'operazione di INSERT e DECREASE-KEY hanno un costo pari a $O(1)$, mentre EXTRACT-MIN ha un costo pari a $O(v)$ dato che siamo costretti a cercare per tutto l'array. Quindi in totale si otterrà una complessità di $O(V^2) + E) = O(V^2)$. Nel caso in cui il grafo sia sufficientemente sparso, in particolare per $E = O(V^2/\lg(V))$ è possibile migliorare l'algoritmo implementando la coda di priorità con un heap-binario. In questo secondo caso, si ha EXTRACT-MIN con costo $O(\lg(V))$, DECREASE-KEY con costo $O(lg(V))$. Quindi il costo totale è $O((V+E)\cdot lg(V))$ ed è pari a $O(E\cdot \lg(V))$ se il grafo è connesso, ovvero tutti i vertici sono raggiungibili dalla sorgente. Considerando la condizione di grafo sparso, il costo migliora rispetto alla coda di priorità realizzata tramite array. Nel terzo caso la coda di priorità può essere implementata con un heap di Fibonacci. In questo caso si ha: EXTRACT-MIN e DECREASE-KEY con costo ammortizzato rispettivamente $O(lg(V))$ e $O(1)$. Quindi il costo totale con heap di Fibonacci è $O(|V|\cdot \log(|V|) + |E|)$.

5. Implementazione

Per implementare l'algoritmo si è utilizzato l'ambiente di sviluppo Java 11 jdk, e l'IDE Visual Studio Code. [2] Il progetto è composto da 7 classi:

- *Main*
- *Node*
- *Edge*
- *Graph*
- *DijkstraAlgorithm*
- *Printer*
- *Test*

5.1 *Main*

Nella classe *Main.java* è presente il metodo *main()*. Fondamentalmente questa classe ha solo due compiti:

- Avviare il programma;
- Ricevere i dati di input.

Infatti possono essere assegnati i parametri di input tramite terminale grazie al metodo privato *getInput()*. I parametri sono:

- *numberNodes*, il numero di nodi da considerare;
- *unitCost*, se settato a true, il costo di ogni arco è pari a 1, se settato a false, il costo di ogni arco sarà casuale;

- *orientedGraph*, se settato a true, gli archi saranno non orientati (bidirezionali), se settato a false, gli archi saranno orientati (unidirezionali);
- *maxCostForEdge*, costo massimo di un arco. Se *unitCost* è false, verrà assegnato verrà assegnato ad ogni arco un costo casuale compreso tra 0 e *maxCostForEdge*;
- *density*, densità degli arco nel grafo, corrisponde al rapporto nodi/archi;
- *verbose*, se settato a true, si vedranno tutte le informazioni dei percorsi trovati, se settato a false, si analizzerà soltanto le prestazioni dell'algoritmo.

Se sono presenti tutti e 6 i parametri allora verranno assegnati, altrimenti verranno utilizzati i parametri definiti dal codice (parametri di default). Dopodichè viene creto e generato un oggetto *Printer*.

5.2 *Node*

La classe *Node* implementa *Comparable* e viene usata per definire l'oggetto nodo. Quindi è presente un costruttore, con il nome del nodo e una *LinkedList<Edge>()* che contiene tutti tutti gli archi che partono dal nodo, realizzando una lista di adiacenza. Sono memorizzati alcuni parametri: il nome del nodo, un booleano per vedere se è stato visitato, un booleano per vedere se è stato connesso (per la creazione del Grafo), una *LinkedList* di archi, la distanza con il nodo source e il nodo predecessore. Oltre ai metodi getter e setter abbiamo un metoso *addNeighbour(Edge edge)* che aggiunge un arco nella *LinkedList* di archi.

5.3 *Edge*

L'oggetto *Edge* rappresenta l'arco tra due nodi. Sono richiesti 3 parametri:

- *weight*, che rappresenta il costo dell'arco;
- *startNode*, nodo da cui parte l'arco;
- *endNode*, nodo su cui arriva l'arco.

Implementa il metodo *toString()*, i getter e i setter.

5.4 *Graph*

L'oggetto *Graph* definisce il grafo, e implementa il metodo *createGraph()*, *firstRun()*, *getAdjNode(int i)*, *toString()*. Il metodo *createGraph()* costruisce il grafo. Dato il numero di nodi, la densità degli archi, il peso degli archi (unitario o casuale fino al valore prestabilito) e la tipoligia di arco(orientato oppure non orientato). Vengono generati degli archi in modo randomico, un nodo causale viene scelto all'inizio e, settato il suo *setConnected* a true, non verrà più scelto come nodo di arrivo da un altro arco. Verranno a sua volta connessi tutti gli altri nodi non connessi fino a che ogni nodo è connesso. Quando questo succede, verrano creati altri archi casuali tra i nodi connessi finchè la densità non viene raggiunta. Questo metodo si preoccupa inoltre di non creare archi doppi. Se viene richiesta la creazione di un grafo con densità troppo elevata, ovvero che non ci sono abbastanza nodi per poter creare il numero richiesto di archi, allora il metodo continuerà il loop. La scelta, quindi viene vincolata a dei valori di densità accettabili. É presente anche una variabile *verbose* che, se settata a true, ci permette di mostrare la creazione del grafo nel terminale. Il metodo *getAdjNode* restituisce il nodo nella posizione i nella lista di adiacenza. [1]

5.5 *DijkstraAlgorithm*

La classe *DijkstraAlgorithm* si occupa del calcolo del percorso minimo dato un nodo sorgente in ingresso attraverso l'algoritmo di Dijkstra. Sono presenti 2 metodi:

- *ShortestP(Node nodeSource)*, inizialmente inizializza la distanza con la sorgente pari a 0, e crea un oggetto *PriorityQueue* (tramite la librearia *java.utils*). La coda di priorità viene memorizzata con un array. É una soluzione efficiente in questo caso perchè si andranno a scansionare tutti i nodi del grafo in quanto l'obiettivo è risolvere il problema *Single-Source Shortest Path* (*SSSP*). Quindi viene iterato un ciclo while finchè la coda di priorità è vuota. Sarà vuota quando finirà di visitare tutti gli archi. Quando trova un percorso più breve da un nodo alla sorgente, viene aggiornato il valore della distanza. Al termine verranno restituiti tutti i percorsi minimi;

- *getShortestP(Node targetNode)*.

5.6 *Printer*

La classe *Printer* è una classe di appoggio tra la classe *Main*, *Test* e il package *algorithm* che contiene la logica dell'algoritmo. La classe *Printer* sostanzialmente

riceve in input i parametri scelti dall'utente, costruisce un oggetto *Graph*, invoca la class *DijkstraAlgorithm* sul grafo creato cronometrando il tempo che ci impiega a risolvere algoritmo. Per ultimo stampa i risultati se il *verbose* è settato a true.

5.7 *Test*

É una classe realizzata per testare le prestazioni dell'algoritmo di Dijkstra. Implementa anch'esso un main. É composta dal main e da un metodo *getInput()*, che servirà per ricevere l'input dall'utente riguardo:

- *numberNodesStart*, numero massimo di nodi con cui verranno effettuati i test. Verranno effettuati i test partendo da 10 ed aumentando di dieci volte tanto ogni volta, fino a che il numero di nodi è minore o uguale a *numberNodesStart*. Ad esempio, se settato a 1000, verrà testato l'algoritmo con un numero di nodi pari a 10, 100, 1000;
- *unitCost*, se settato a true, il costo di ogni arco è pari a 1, se settato a false, il costo di ogni arco sarà casuale;
- *orientedGraph*, se settato a true, gli archi saranno non orientati (bidirezionali), se settato a false, gli archi saranno orientati (unidirezionali);
- *maxCostForEdge*, costo massimo di un arco. Se *unitCost* è false, verrà assegnato verrà assegnato ad ogni arco un costo casuale compreso tra 0 e *maxCostForEdge*;
- *densityStart*, densità degli arco nel grafo, corrisponde al rapporto nodi/archi;
- *verbose*, se settato a true, si vedranno tutte le informazioni dei percorsi trovati, se settato a false, si analizzerà soltanto le prestazioni dell'algoritmo.

Se non si desidera scegliere i parametri, si possono comunque utilizzare quelli di default. Questa classe di test eseguirà il proprio main per diverse configurazioni di input, così da ottenere dei test velocemente e facilitandone il processo.

6. Dati Sperimentali

La tabella seguente rappresenta i risultati dei test condotti per valutare la bontà dell'algoritmo di Dijkstra al variare della densità e della grandezza del grafo. I risultati sono stati ottenuti eseguendo il main della classe *Testing* con un massimo di nodi pari a 10000, con gli archi unidirezionali (quindi con un grafico orientato), e con densità variabile da 1 a 4, aumentando di 0,2 in 0,2. Per ogni configurazione sono stati eseguiti 3 test identici così da minimizzare le imprecisioni dovute da cali di prestazioni, o a grafi particolarmente fortunati o sfortunati da calcolare. Se ad esempio il nodo source non è connesso, il calcolo dei percorsi minimi sarà molto rapido. anche se il numero di nodi e la densità sono elevati. Nella colonna Average Time, è stata svolta una media aritmetica tra i 3 test con lo stesso input. Dai dati ottenuti sono stati realizzati alcuni grafici:

- Il primo grafico mostra l'andamento dei tempi di esecuzione in millisecondi all'aumentare della densità, ovvero il rapporto tra nodi e archi. La densità va da 1.0 fino ad 4.0 (ovvero fino a quando gli archi sono 4 volte in più dei nodi). É abbastanza evidente che all'aumentare della densità, il tempo di esecuzione cresce con un ritmo più lento di $O|V|^2 + E$. É interessante notare che anche considerando un numero molto importante di nodi, se la densità rimane bassa, il tempo di esecuzione sarà comunque basso. Quindi, per analizzare al meglio il tempo di esecuzione all'aumentare dei nodi è stato realizzato un secondo grafico.

- Il secondo grafico mostra l'andamento dei tempi di esecuzione in millisecondi all'aumentare dei nodi. Ogni linea corrisponde ad un valore di densità. Importante sottolineare che l'asse delle ordinate è logaritmico. Essendo la scala logaritmica, si nota dal grafico che, per il numero di nodi basso, resta pressocchè invariato per diversa densità. Quando il numero di nodi aumenta, invece, il tempo di esecuzione inizia a differenziarsi. Per valori alti, infatti, i tempi di esecuzione sembrano aumentare linearmente sulla scala logaritmica. Il che significa che l'aumento è pressocchè logaritmico.

NODES:	DENSITY:	TIME1(ns):	TIME2(ns):	TIME3(ns):	AVERAGETIME:	AVERAGE TIME for each density(ms):
10	1.0	707400	63400	55200	275333	1392,49
100	1.0	233200	219700	199400	217433	
1000	1.0	1435300	1013200	892000	1113500	
10000	1.0	4615500	4615500	2660100	3963700	
10	1.2	71600	90200	74300	78700	1926,62
100	1.2	302900	249900	164400	239067	
1000	1.2	942500	926500	991100	953367	
10000	1.2	6632100	6420900	6253000	6435333	
10	1.4	79700	67800	96100	81200	2160,94
100	1.4	191800	237800	247200	225600	
1000	1.4	1024800	1065700	1072500	1054333	
10000	1.4	7044200	7283100	7520600	7282633	
10	1.6	82800	98600	125500	102300	2557,28
100	1.6	205900	198600	176500	193667	
1000	1.6	1153300	637400	478300	756333	
10000	1.6	8488300	10863200	8179000	9176833	
10	1.8	130300	114500	90900	111900	2634,58
100	1.8	185000	199500	174600	186367	
1000	1.8	426900	466300	647000	513400	
10000	1.8	10005000	10145900	9029100	9726667	
10	2.0	88100	105800	97400	97100	3144,63
100	2.0	186500	190200	184800	187167	
1000	2.0	491600	454300	482000	475967	
10000	2.0	13191100	10496600	11767200	11818300	
10	2.2	103900	105800	97400	102367	3635,83
100	2.2	198300	224600	192900	205267	
1000	2.2	518300	1009300	422600	650067	
10000	2.2	14139200	13432100	13185500	13585600	
10	2.4	101100	101100	160200	120800	3186,39
100	2.4	197000	224300	233100	218133	
1000	2.4	452200	390800	406800	416600	
10000	2.4	11600100	11857800	12512200	11990033	
10	2.6	131100	138800	96600	122167	3465,53
100	2.6	225700	239000	239200	234633	
1000	2.6	512300	445700	583800	513933	
10000	2.6	12954600	13139900	12879700	12991400	
10	2.8	93900	57100	58900	69967	4091,15
100	2.8	220500	237800	278800	245700	
1000	2.8	485800	350700	417000	417833	
10000	2.8	14192600	19587100	13113600	15631100	
10	3.0	60100	74300	54400	62933	4932,52
100	3.0	238500	244300	275000	252600	
1000	3.0	423300	353000	560200	445500	
10000	3.0	22061700	17193600	17651800	18969033	
10	3.2	69600	69300	110600	83167	3869,35
100	3.2	287800	227000	231200	248667	
1000	3.2	351900	449500	610100	470500	
10000	3.2	15330700	15007100	13687400	14675067	
10	3.4	61400	58100	86000	68500	

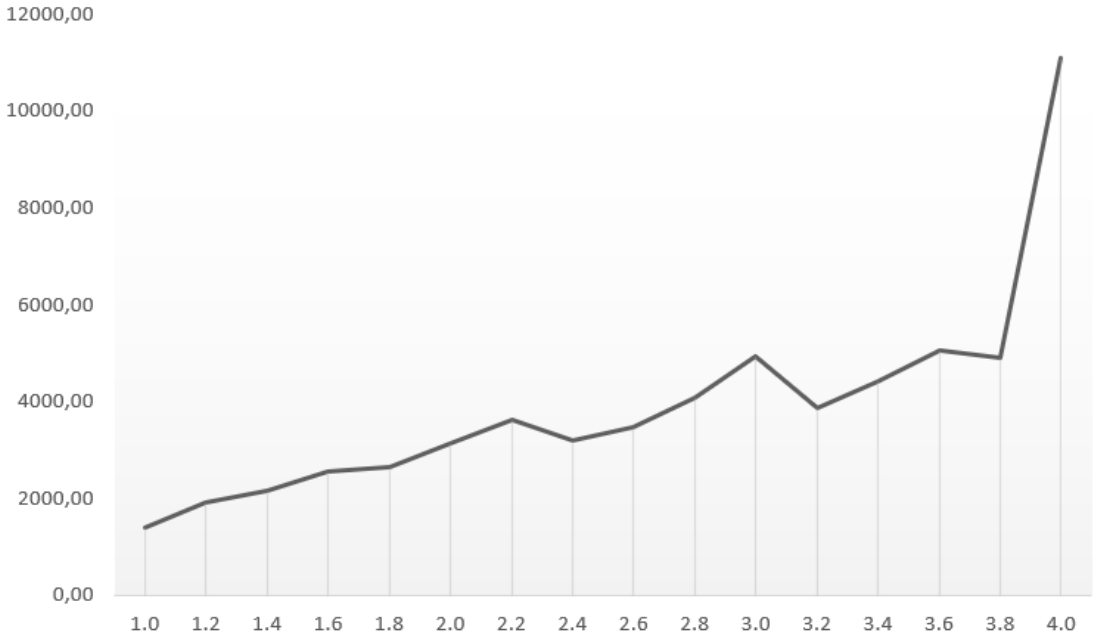

Figura 6.1

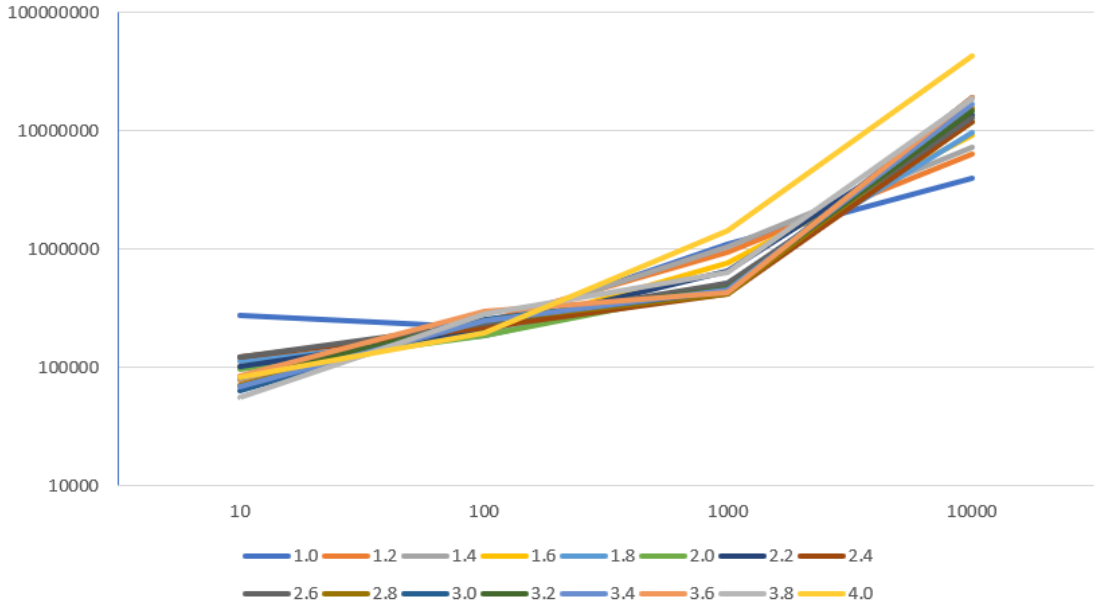

Figura 6.2

7. Bibliografia

[1] java linkedlist. disponibile su https://docs.oracle.com/javase/7/docs/api/.

[2] java node. disponibile su https://docs.oracle.com/javase/7/docs/api/.

[3] Liste di adiacenza. disponibile su https://www.programiz.com/dsa/graph-adjacency-list.

[4] Algoritmo di dijkstra, Apr 2023. disponibile su https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra.

[5] Dijkstra, May 2023. disponibile su https://en.wikipedia.org/wiki/Edsger_W._Dijkstra.

[6] Grafi, May 2023. disponibile su https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

[7] Lista di adiacenza, May 2023. disponibile su https://it.wikipedia.org/wiki/Lista_di_adiacenza.

[8] Problema del cammino minimo, Jun 2023. disponibile su https://en.wikipedia.org/wiki/Shortest_path_problem.

[9] Emilio Digiacomo. 16- grafi, Apr 2023.

[10] Emilio Digiacomo. 18- cammini minimi, Apr 2023.